Aim:
Finding lane in given videos using techniques studied in the lecture including camera calibration, undistorting, thresholding, perspective transform and identifying lanes using sliding window search and previous lane history.

Final Pipeline:
The pipeline is implemented in pipeline.py and includes following steps:

- Undistorting the image
- Applying thresholds to get a binary image which highlights lanes
- Applying top view transform to get birds eye view of the lane
- Identifying lane pixels, radius of curvature, distance from center
- Projecting identified lane, radius and distance from center back onto image

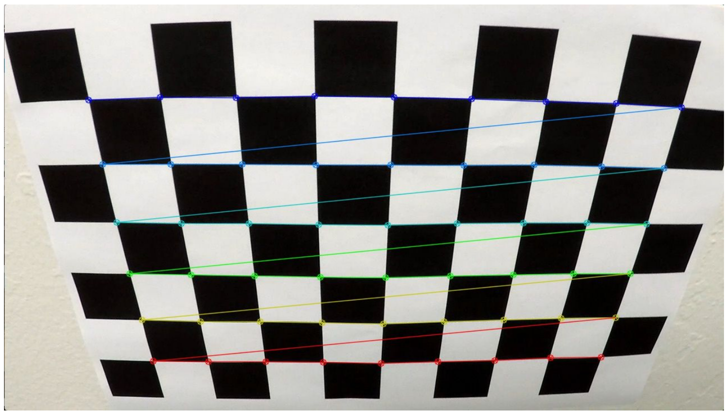
Output videos:
- project_video.mp4
- Challenge_video.mp4

Output images:
- All output images used are stored in output_images folder, some of which are included in this report in respective sections.

Following sections describe various stages of the pipeline.

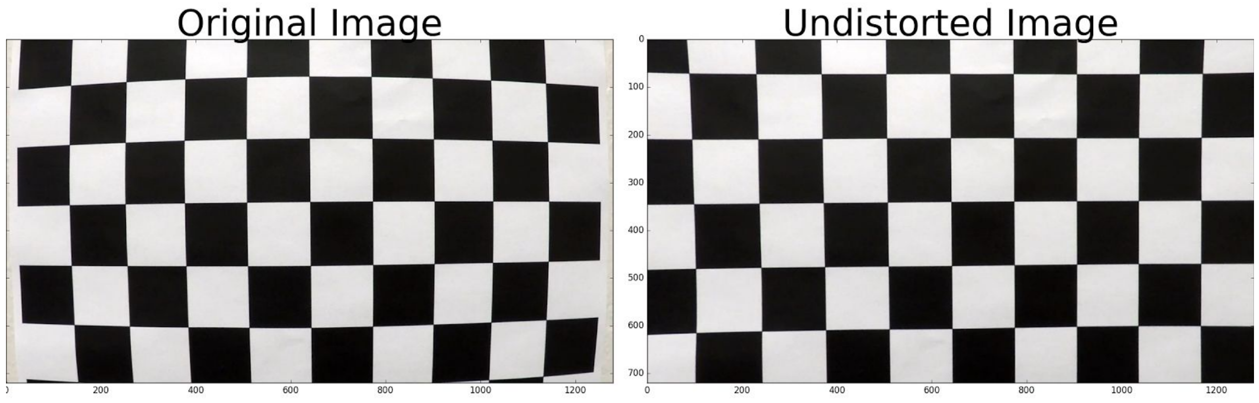
Camera Calibration:
Camera was calibrated using images in camera_cal using opencv function findChessboardCorners and calibrateCamera (Relevant code in **get_dist_mtx.py**). The code dumps the camera matrices in distortion.p

which is later used to undistort images (undistort.py). Example images:



Distortion correction:

The relevant code is in **undistort.py** and uses opencv function cv2.undistort and matrices calculated during camera calibration



Thresholding:

- Code in threshold.py
- Color thresholding based on S, R and L channels
    - High R threshold is good for some of the white lines
- Region of interest thresholding
- Absolute x-gradient and direction holding
    - I tried various kernel sizes. High kernel size gave great result for nearby portions of the lane, but the far ends got so noisier that it interfered with lane detection algorithm. Small size did well at far end, but gave lesser pixels for near ends.

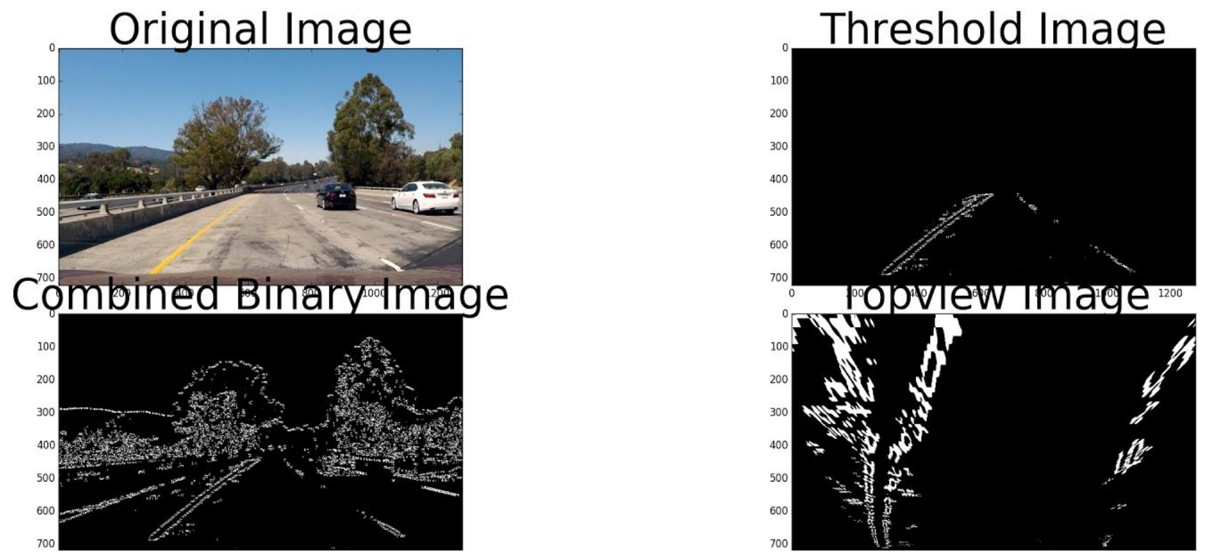
Overall:

```
gray = cv2.cvtColor(oimg, cv2.COLOR_RGB2GRAY) # img is read
via mpimg
abs_binary = abs_grad_threshold(gray, 'x', 40, 120, 5) # abs
x-gradient
d_binary = dir_grad_threshold(gray, 1.0, 1.5, 7) # Dir gradient
gray_grad = np.zeros_like(abs_binary)
cond = ((d_binary == 1) & (abs_binary == 1))
gray_grad[cond] = 1

r_binary = r_threshold(oimg, 225, 255) # R channel
s_binary = s_threshold(oimg, 80, 255) # S channel
l_binary = l_threshold(oimg, 200, 255) # L channel

color_thresh_binary = grad_threshold(r_binary | s_binary |
l_binary)
shape = color_thresh_binary.shape
h, w = shape[0], shape[1]
vertices = np.array([[(150, h), (570, 440),
                      (715, 440), (1150, h)]], dtype=np.int32)
timg = (color_thresh_binary | gray_grad)
roi = region_of_interest(timg, vertices)
return r_binary, color_thresh_binary, gray_grad, roi
```

Grad thresholding:

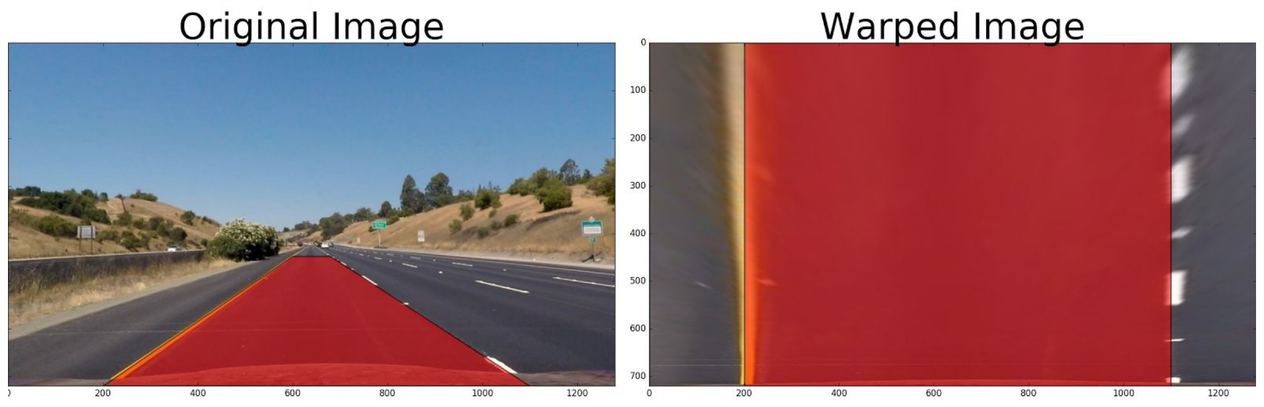
```
abs_binary = abs_grad_threshold(img, 'x', 20, 100, 7) # abs
x-gradient
d_binary = dir_grad_threshold(img, 0.7, 1.5, 7) # Dir gradient
combined_binary = np.zeros_like(abs_binary)
cond = ((d_binary == 1) & (abs_binary == 1))
combined_binary[cond] = 1
return combined_binary
```

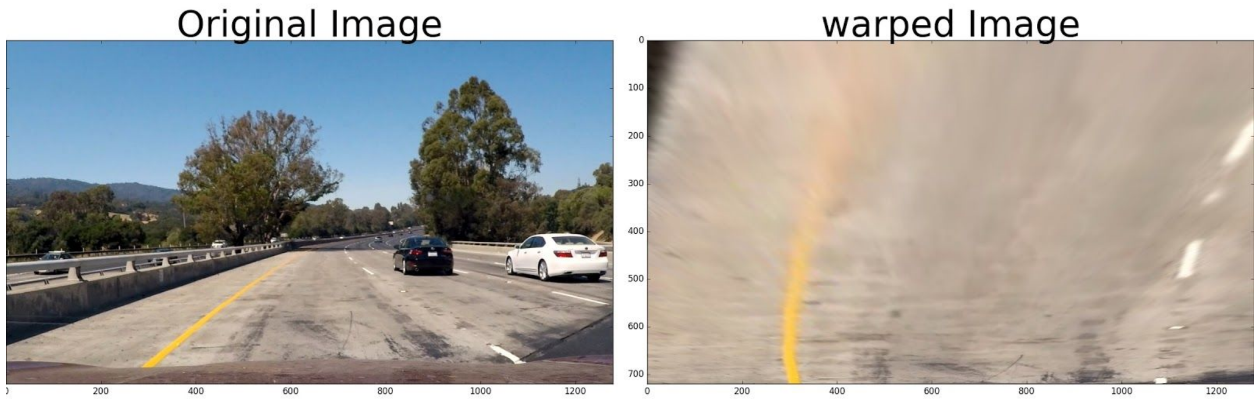
Sample image depicting thresholding debug process (output_images/threshold_test1.jpg):



Transformation:

Transformation matrices for top view are calculated in **calculate_top_view_transform.py** and dumped in transform.p which are later used in **topview.py** to transform any image. The points for calculating transformation are chosen manually from straight_lines1.jpg image which is provided in project repo. Image example after transformation:





Lane Identification:

Lane identification is done using sliding window algorithm as described in lecture using histogram method. In case a previous high confidence lane is known, targeted lane search is done. The relevant code is in sliding_window.py and find_lane.py. Sliding_window implements the algorithm while find_lane keeps track of previous lane locations, filtering new lanes on basis of sanity checks and smoothening the found lane curve.

LaneFinder (find_lane.py):

- **Lane Finding**:
    - It keeps track of how many frames ago a high confidence lane was found,
    - and on basis of that either does a full sliding window search or a targeted search in region of last lane.
- **Sanity checks**:
    - Min pixel check: Both lines should have a minimum number of pixels detected,
    - Proximity to last lines: each of the left and right lines should not be too much away from last lines reported.
    - Parallel check: Both lines should also be reasonably parallel to each other.

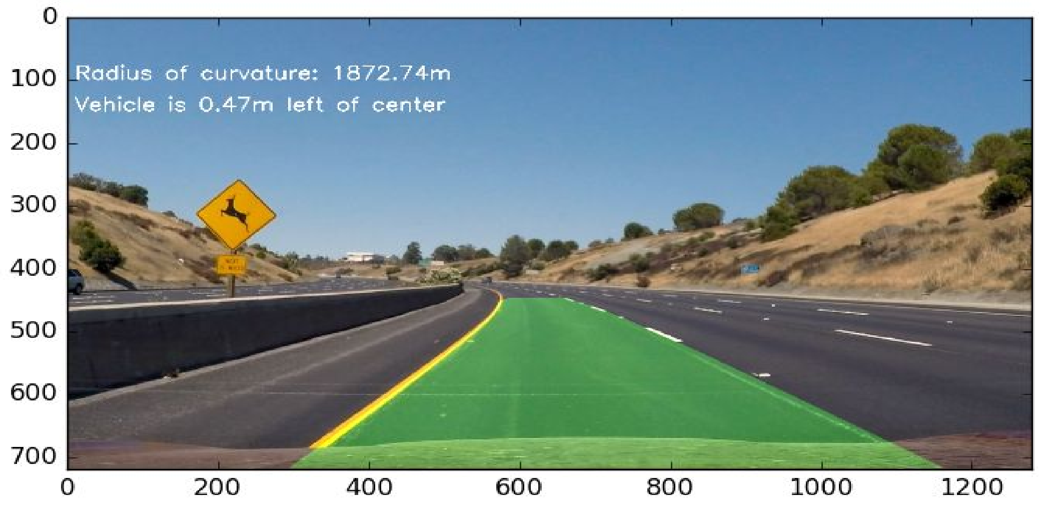
        - If all conditions are satisfied, the lane data is used, otherwise only left line or right line data is used if any of them is individually valid.
    - **Smoothening**: The reported lane is averaged over last few lane positions

Radius of curvature and vehicle position:

It’s been calculated in **sliding_window.py**. Relevant code:

```
def roc(y_eval, left_fit, right_fit):
   # Define conversions in x and y from pixels space to meters
   ym_per_pix = 30/720 # meters per pixel in y dimension

   # Calculate the radii of curvature
   left_curverad = ((1 + (2 * left_fit[0] * y_eval * ym_per_pix + left_fit[1])*
*2)**1.5) / np.absolute(2 * left_fit[0])
   right_curverad = ((1 + (2 * right_fit[0] * y_eval * ym_per_pix + right_fit[1
])**2)**1.5) / np.absolute(2 * right_fit[0])
   # Now our radius of curvature is in meters
   return left_curverad, right_curverad

def dist_from_center(binary_warped, left_fit, right_fit):
   # assuming lane center should be at center of image
   xm_per_pix = 3.7/700 # meters per pixel in x dimension

   y = binary_warped.shape[0] - 1
   xl = left_fit[0]*y**2 + left_fit[1]*y + left_fit[2]
   xr = right_fit[0]*y**2 + right_fit[1]*y + right_fit[2]
   lane_center = (xl + xr) / 2
   img_center = binary_warped.shape[1] / 2
   return (lane_center - img_center) * xm_per_pix
```

Example image from project_video:


Radius of curvature: 1872.74m
Vehicle is 0.47m left of center

Discussion:

- **Problems faced**: Majority problem was in thresholding the images for challenge_video to get the right lanes as the colors were on the dull side and the road has other lines which could be detected as lanes and it had features which were being picked by lower S threshold. Looking too far ahead in the lane also poses problems as transform of far ends of the lane gets noisier. In case only few pixels are detected in the lanes especially towards the far end, the polynomial fit of degree 2 can really diverge from frame to frame. Results on challenge_video were not as good, as the left shoulder width was quite less, thus confusing the algorithm
- **Possible failure cases:** The pipeline is geared towards smooth lane transitions, so anywhere, where there are sharper turns than the thresholds defined, it would fail. It also assumes a certain mix-max range for lane width which might not hold at all places. Old lane

markings on the road and sharper turns seems to be the biggest problems here. Also, if the vehicle goes a bit off track and one of the lines ends up in the middle of the image, this might break

- **Possible improvements:** One obvious improvement could be to divide the image into sections along y axis and have individual fits, which would help with sharper turns. It might also help in better transitioning/smoothening ability from frame to frame. Lane detection algorithm can also be improved to handle off-center cases better.